## **Знакомство с рекурсией.**

Рекурсия – это прием программирования, при котором программа вызывает саму себя либо непосредственно, либо косвенно.

Как правило, неопытный программист, узнав про рекурсию, испытывает легкое недоумение. Первая мысль – это бессмысленно!!! Такой ряд вызовов превратиться в вечный цикл, похожий на змею, которая съела сама себя, или приведет к ошибке на этапе выполнения, когда программа поглотит все ресурсы памяти.

Однако рекурсия – это превосходный инструмент, который при умелом и правильном использовании поможет программисту решить множество сложных задач.

### **Пример на рекурсию**

Исторически сложилось так, что в качестве первого примера на рекурсию почти всегда приводят пример вычисления факториала.

Что ж, не будем нарушать традиций.

Для начала, вспомним, что такое факториал. Обозначается факториал восклицательным знаком «!» и вычисляется следующим образом:

|  |
| --- |
| N! = 1 \* 2 \* 3 \* ... \* N |

Другими словами, факториал представляет собой произведение натуральных чисел от 1 до N включительно. Исходя из вышеописанной формулы, можно обратить внимание на следующую закономерность:

|  |
| --- |
| N! = N \* (N-1)! |

Ура!!! Мы можем найти факториал через сам факториал! Вот здесь мы и попадаемся в ловушку. Наша находка, на первый взгляд, абсолютно бесполезна, ведь неизвестное понятие определяется через такое же неизвестное понятие, и получается бесконечный цикл. Выход из данной ситуации сразу же будет найден, если добавить к определению факториала следующий факт:

1! = 1

Теперь мы можем себе позволить вычислить значение факториала любого числа. Попробуем, например, получить 5!, несколько раз применив формулу N! = N \* (N-1)! и один раз формулу 1! = 1:

|  |
| --- |
| 5! = 5 \* 4! = 5 \* 4 \* 3! = 5 \* 4 \* 3 \* 2! = 5 \* 4 \* 3 \* 2 \* 1! = 5 \* 4 \* 3 \* 2 \* 1 |

Как же будет выглядеть данный алгоритм, если перенести его на язык С? Давайте, попробуем реализовать рекурсивную функцию:

|  |
| --- |
| #include <iostream> using namespace std;  long int Fact(long int N) {  // если произведена попытка вычислить факториал нуля  // или единицы - вернуть 1,  // именно здесь произведется выход из рекурсии   if (N == 1 || N == 0) return 1;  // любое другое число вызывает функцию заново с формулой N-1  else return N \* Fact(N-1); }   void main() {  long number=5;  //первый вызов рекурсивной функции  long result=Fact(number);  cout<<"Result "<<number<<"! is - "<<result<<"\n"; } |

Как видите, всё не так уж сложно. Для более детального понимания примера рекомендуем скопировать текст программы в Visual Studio и пошагово пройтись по коду отладчиком.

## **Рекурсии или итерации?**

Изучив предыдущий раздел урока – вы наверняка задались вопросом: а зачем нужна рекурсия? Ведь, реализовать вычисление факториала можно и с помощью итераций и это совсем не сложно:

|  |
| --- |
| #include <iostream> using namespace std;  long int Fact2(long int N) {  long int F = 1;  //цикл осуществляет подсчет факториала  for (long int i=2; i<=N; i++)  F \*= i;  return F; }  void main() {  long number=5;  long result=Fact2(number);  cout<<"Result "<<number<<"! is - "<<result<<"\n"; } |

Такой алгоритм, наверное, будет более естественным для программистов. На самом деле, это не совсем так. С точки зрения теории, любой алгоритм, который можно реализовать рекурсивно, совершенно спокойно реализуется итеративно. Мы только что в этом убедились.

Однако это не совсем так. Рекурсия производит вычисления гораздо медленнее, чем итерация. Кроме того, рекурсия потребляет намного больше оперативной памяти в момент своей работы.

Значит ли это, что рекурсия бесполезна? Ни в коем случае!!! Существует ряд задач, для которых рекурсивное решение тонко и красиво, а итеративное – сложно, громоздко и неестественно. Ваша задача, в данном случае – научиться, не только оперировать рекурсией и итерацией, но и интуитивно выбирать, какой из подходов применять в конкретном случае. От себя можем сказать, что лучшее применение рекурсии – это решение задач, для которых свойственна следующая черта: решение задачи сводится к решению таких же задач, но меньшей размерности и, следовательно, гораздо легче разрешаемых.

Удачи Вам на данном поприще!!! Как говорится: «Что бы понять рекурсию, надо просто понять рекурсию».

## **Быстрая сортировка.**

"Быстрая сортировка" - была разработана около 40 лет назад и является наиболее широко применяемым и в принципе самым эффективным алгоритмом. Метод основан на разделении массива на части. Общая схема такова:

1. Из массива выбирается некоторый опорный элемент a[i].

2. Запускается функция разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], слева от него, а все ключи, большие, либо равные a[i] - справа, теперь массив состоит из двух частей, причем элементы левой меньше элементов правой.

3. Если в подмассивасивах более двух элементов, рекурсивно запускаем для них ту же функцию.

4. В конце получится полностью отсортированная последовательность.

Рассмотрим алгоритм более детально.

### **Делим массив пополам.**

Входные данные: массив a[0]...a[N] и элемент p, по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.

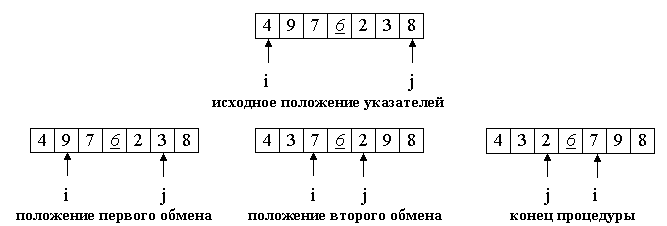
2. Будем двигать указатель i с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p.

3. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.

4. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i,j по тем же правилам.

5. Повторяем шаг 3, пока i <= j.

Рассмотрим рисунок, где опорный элемент p = a[3].



Массив разделился на две части: все элементы левой меньше либо равны p, все элементы правой - больше, либо равны p.

### **Пример программы.**

|  |
| --- |
| #include <iostream> #include <stdlib.h> #include <time.h> using namespace std;  template <typename T> void quickSortR(T a[], int B, int E) {  long i = B, j = E;   T temp, p;  p = a[(B+E)/2];  do{  while ( a[i] < p ) i++;  while ( a[j] > p ) j--;  if (i <= j){  temp = a[i];  a[i] = a[j];  a[j] = temp;  i++;  j--;  }  }while ( i<=j );  if(B<j)quickSortR(a, B, j);  if(i<E)quickSortR(a, i, E); }  void main(){  srand(time(NULL));  const long SIZE=10;  int ar[SIZE];    // до сортировки  for(int i=0;i<SIZE;i++){  ar[i]=rand()%100;  cout<<ar[i]<<"\t";  }  cout<<"\n\n";  quickSortR(ar,0,SIZE-1);   // после сортировки  for(int i=0;i<SIZE;i++){  cout<<ar[i]<<"\t";  }  cout<<"\n\n"; } |

##### **Алгоритм рекурсии.**

1. Выбрать опорный элемент p - середину массива

2. Разделить массив по этому элементу

3. Если подмассив слева от p содержит более одного элемента, вызвать quickSortR для него.

4. Если подмассив справа от p содержит более одного элемента, вызвать quickSortR для него.

## **Двоичный поиск.**

В прошлом уроке мы рассмотрели алгоритм линейного поиска, однако это не единственная возможность организовать поиск в массиве. Если у нас есть массив, содержащий упорядоченную последовательность данных, то, в данном случае, очень эффективен двоичный поиск.

### **Теория двоичного поиска.**

Предположим, что переменные Lb и Ub содержат, соответственно, левую и правую границы отрезка массива, где находится нужный нам элемент. Поиск мы всегда будем начинать с анализа среднего элемента отрезка массива. Если искомое значение меньше среднего элемента, мы переходим к поиску в верхней половине отрезка, где все элементы меньше только что проверенного. Другими словами, значением Ub становится (M (средний элемент) – 1) и на следующей итерации мы работаем с половиной массива. Таким образом, в результате каждой проверки мы вдвое сужаем область поиска. Так, в нашем примере, после первой итерации область поиска – всего лишь три элемента, после второй остается всего лишь один элемент. Таким образом, если длина массива равна 6, нам достаточно трех итераций, чтобы найти нужное число.

### **Пример реализации.**

|  |
| --- |
| #include <iostream> #include <stdlib.h> #include <time.h>  using namespace std;  int BinarySearch (int A[], int Lb, int Ub, int Key) {  int M;  while(1){  M = (Lb + Ub)/2;  if (Key < A[M])  Ub = M - 1;  else if (Key > A[M])  Lb = M + 1;  else  return M;  if (Lb > Ub)  return -1;  } }  void main(){  srand(time(NULL));  const long SIZE=10;  int ar[SIZE];  int key,ind;    // до сортировки  for(int i=0;i<SIZE;i++){  ar[i]=rand()%100;  cout<<ar[i]<<"\t";  }  cout<<"\n\n";  cout<<"Enter any digit:";  cin>>key;  ind=BinarySearch(ar,0,SIZE,key);  cout<<"Index - "<<ind<<"\t";   cout<<"\n\n"; } |

Двоичный поиск - очень мощный метод. Посудите сами: например, длина массива равна 1023, после первого сравнения область сужается до 11 элементов, а после второй - до 255. Легко посчитать, что для поиска в массиве из 1023 элементов достаточно 10 сравнений.